PUC-Rio - Certificação Digital Nº 0521495/CA

1 Introduction

1.1 Motivation and Objectives

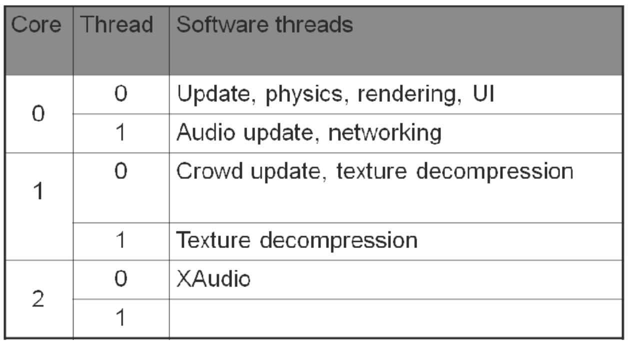
Current generation video game consoles (Xbox 360 and Playstation 3) have multiple processors allowing parallel execution of the game tasks [Xbox360Hardware], [CellMicroprocessor]. However, many games are not reaching their full potential because they are not properly threaded for multi-core architectures. In order to appropriately use this type of hardware, it is necessary to change the usual ways of solving problems in game programming. Furthermore, game engine architectures must evolve towards more flexible and efficient ways of dividing the workload between the processing cores.

Game engine is a class of software framework extensively used by the game industry. There are many ways of dividing game engine tasks between multiple cores of a computer. One way is simply taking a task from a single threaded game engine and placing it to run on another core. The performance increase will be the result of how much parallelism the task separation allows and the amount of communication overhead.

Figure 1.1 shows how a game on the Xbox 360 divides its tasks between the different cores of the processor. The processor of the Xbox 360 has 3 symmetrical cores with two hardware threads per core. The threads in this figure reveal that the engine has its main processing done in a single core. The other cores are used for graphical complementary tasks and audio. This is an unbalanced task division of a game, where a single core is responsible for the most important tasks. Figure 1.2 presents a game with a more balanced task division than the one found in figure 1.1.

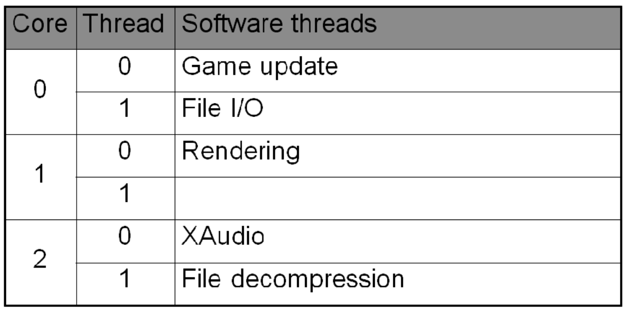
It is possible to divide the workload of a game on a multicore processor not only by placing different tasks on different cores. It is also possible to implement parallel versions of game algorithms in several game activities, such as graphics, physics, collision detection and artificial intelligence.

However, despite the importance of parallelism for the game industry, there are few related academic works available. The literature on parallel



Core	Thread	Software threads
0	0	Update, physics, rendering, UI
	1	Audio update, networking
1	0	Crowd update, texture decompression
	1	Texture decompression
2	0	XAudio
	1	

Figure 1.1: Project Gotham Racing Threads (source:[Dawson06])



Core	Thread	Software threads
0	0	Game update
	1	File I/O
1	0	Rendering
	1	
2	0	XAudio
	1	File decompression

Figure 1.2: Kameo's Threads - a more balanced task division (source:[Dawson06])

game engine techniques and architectures is scarce and game programmers usually consult classic references [Grama03], [Wilkinson05]. Furthermore, most of the work on parallel rendering is focused on PC clusters and grids, while the literature on parallel rendering for multicore systems is rare.

This work has the objective of contributing to the research of parallel game engine technology by looking at the different ways a game programmer can organize and improve the parallel game engine tasks. Firstly, we look at this task division on an architectural level. Then we present parallel algorithms on computer graphics and collision detection, which are standard targets for parallelization in games.

The present thesis is about strategies and algorithms for parallel game engines. The computer graphics algorithms that will be analyzed will not include any rendering and, as such, there will be no 3D rendered scenes in this work. The reader should visit the project web page (see section 1.3) to find some systems based on the present work.

The solutions presented on this thesis are all based on multicore CPUs and don't use CUDA [Kirk10]. CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA that allow programmers to use highly parallel GPUs (Graphical Processing Units)

PUC-Rio - Certificação Digital Nº 0521495/CA

PUC-Rio - Certificação Digital Nº 0521495/CA

and a C like language to solve general problems. CUDA is an excelent solution to solving parallel problems that don't required a lot of branching (such as traversing a space partitioning data structure). The reason we didn't researched CUDA solutions was to have more time to find better architectures and algorithms for multicore CPUs.

1.2 The Organization of the thesis

Chapter 2 of this thesis deals with parallel game engine architectures and shows different forms to divide game tasks. As far as we are aware, there is no reference in the literature to a comprehensive analysis between different parallel game engine architectures. Chapter 2 contributes to narrow this gap in the literature. A partial analysis on this subject can be found in works by Monkkonen [Monkkonen06] and Lake [Lake05].

Sometimes, in order to achieve scalability, it is necessary to transform standard game engine algorithms into parallel algorithms. Chapter 3 and 4 deal with parallel versions of standard algorithms in games for computer graphics and collision detection.

Octree is a classic data structure generally used in games for optimizing scene rendering. An Octree recursively divides space into eight chunks until a certain level is reached (figure 1.3). In computer graphics, this data structure is used for quickly eliminating parts of the scene by testing the intersection of the camera frustum against the octree nodes. If a node is not intersected by the frustum, its children will also not be intersected.

Chapter 3 provides a parallel implementation of octree culling. It also presents how to sort resources in parallel in order to accelerate the rendering process. Sorting resources before rendering reduces the amount of resource changes requested by the graphics device. Since changing the render state of the graphics device can be an expensive operation [ResourceChangeCost], reducing the number of render state changes will improve rendering performance.

Collision detection systems usually work in two steps. The first step is the broad phase of the collision detection, which detects objects that are close to each other. The second step is the narrow phase of the collision detection, where the objects that are close to each other are tested for collision. The first step is necessary to reduce the amount of tests in the narrow phase. Without the broad phase, we would have to perform O(n^2) tests on all objects of the scene. By using the broad phase, the complexity of O(n^2) occurs only for the groups of objects that are close to each other.

One way to efficiently find objects that are close to each other is to divide the world into a grid of equal sized cells. Chapter 4 provides algorithms

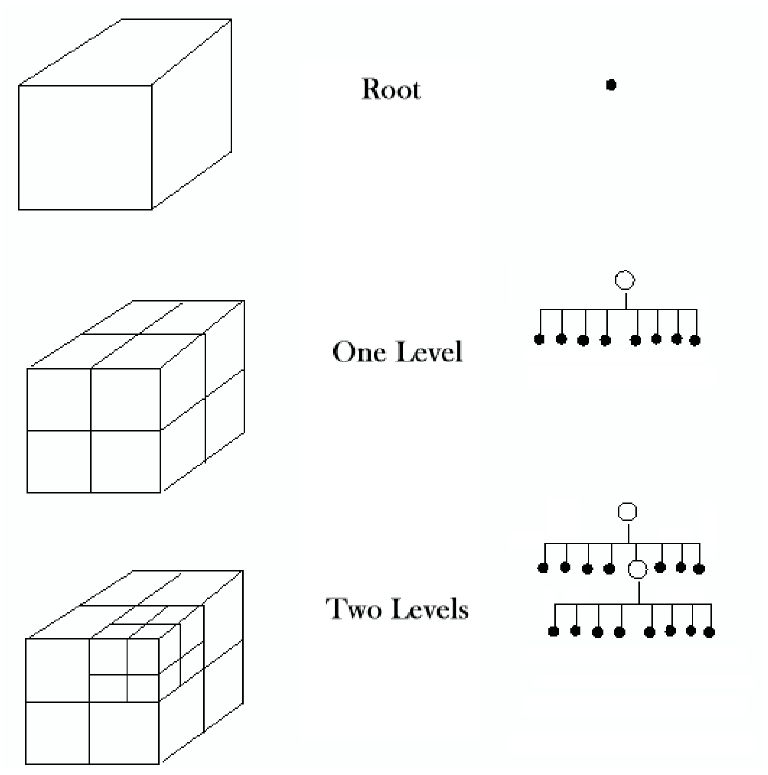


Figure 1.3: An octree

for implementing a parallel hierarchical grid. This chapter also provides information on how to balance the workload of O(n^2) tests from a group of objects that need to be tested for collision.

Chapter 5 presents conclusions, contributions and future works. This thesis brings contributions on three areas:

1. On chapter 2, it proposes a game engine architecture called Fully Parallel Architecture that has greater scalability that the other architectures found in the literature.

2. On chapter 3, it proposes a parallel version of the classical Octree Frustum Culling algorithm. The thesis also proposes, on this chapter, a method for parallel resource sorting.

3. On chapter 4, it proposes a parallel method for computing Broad Phase Collision Detection using Uniform Hierarquical Grids. It is also proposed, on this chapter, a balanced method for Parallel Narrow Phase Collision Detection.

PUC-Rio - Certificação Digital Nº 0521495/CA

1.3 Programming Aspects

The present thesis has a greater focus on algorithms and programming than realistic scenes rendered in real time. Therefore, it is necessary that we clarify codes and programming techniques formats before we start presenting the next chapters.

The pseudocodes presented in this thesis adopt the following conventions to become more readable and concise:

1. standard constructs and keywords are marked in bold face, such as **if, else, for, while, true,** ... .
2. Blocks of statements are delimited by the word **end** (**end if**,**end for**, ...)
3. variable names are in italics (to avoid confusion with free text).
4. comments appear between symbols { and } such as {this is a comment}
5. Attributes are denoted by a dot (.) and indirections are indicated by an arrow (→). For example: $box.min$ and $node \rightarrow children[i]$.
6. Shorthand operators are used (+=, -=, ...).

The task of writing algorithms for multicore systems is complex and requires complete domain on the programming language being used. Pseudo-codes cannot present most of the details and complex solutions used in the real code. For the sake of comprehensibility, all the source code written for this thesis can be examined in the following Web page:

`www.icad.puc-rio.br/projects/~multicoregames`

PUC-Rio - Certificação Digital Nº 0521495/CA